26. Android Touch and Multi-touch Event Handling

Most Android based devices use a touch screen as the primary interface between user and device. The previous chapter introduced the mechanism by which a touch on the screen translates into an action within a running Android application. There is, however, much more to touch event handling than responding to a single finger tap on a view object. Most Android devices can, for example, detect more than one touch at a time. Nor are touches limited to a single point on the device display. Touches can, of course, be dynamic as the user slides one or more points of contact across the surface of the screen.

Touches can also be interpreted by an application as a *gesture*. Consider, for example, that a horizontal swipe is typically used to turn the page of an eBook, or how a pinching motion can be used to zoom in and out of an image displayed on the screen.

The objective of this chapter is to highlight the handling of touches that involve motion and to explore the concept of intercepting multiple concurrent touches. The topic of identifying distinct gestures will be covered in the next chapter.

26.1 Intercepting Touch Events

Touch events can be intercepted by a view object through the registration of an *onTouchListener* event listener and the implementation of the corresponding *onTouch()* callback method. The following code, for example, ensures that any touches on a ConstraintLayout view instance named *myLayout* result in a call to the *onTouch()* method:

```
myLayout.setOnTouchListener(
        new ConstraintLayout.OnTouchListener() {
                public boolean onTouch(View v, MotionEvent m) {
                        // Perform tasks here
                        return true;
                }
        }
);
```

As indicated in the code example, the *onTouch()* callback is required to return a Boolean value indicating to the Android runtime system whether or not the event should be passed on to other event listeners registered on the same view or discarded. The method is passed both a reference to the view on which the event was triggered and an object of type *MotionEvent*.

26.2 The MotionEvent Object

The MotionEvent object passed through to the *onTouch()* callback method is the key to obtaining information about the event. Information contained within the object includes the location of the touch within the view and the type of action performed. The MotionEvent object is also the key to handling multiple touches.

26.3 Understanding Touch Actions

An important aspect of touch event handling involves being able to identify the type of action performed by the user. The type of action associated with an event can be obtained by making a call to the *getActionMasked()* method of the MotionEvent object which was passed through to the *onTouch()* callback method. When the first touch on a view occurs, the MotionEvent object will contain an action type of ACTION_DOWN together with the coordinates of the touch. When that touch is lifted from the screen, an ACTION_UP event is generated. Any motion of the touch between the ACTION_DOWN and ACTION_UP events will be represented by ACTION_MOVE events.

When more than one touch is performed simultaneously on a view, the touches are referred to as *pointers*. In a multi-touch scenario, pointers begin and end with event actions of type ACTION_POINTER_DOWN and ACTION_POINTER_UP respectively. In order to identify the index of the pointer that triggered the event, the *getActionIndex()* callback method of the MotionEvent object must be called.

26.4 **Handling Multiple Touches**

The chapter entitled *An Overview and Example of Android Event Handling* began exploring event handling within the narrow context of a single touch event. In practice, most Android devices possess the ability to respond to multiple consecutive touches (though it is important to note that the number of simultaneous touches that can be detected varies depending on the device).

As previously discussed, each touch in a multi-touch situation is considered by the Android framework to be a *pointer*. Each pointer, in turn, is referenced by an *index* value and assigned an *ID*. The current number of pointers can be obtained via a call to the *getPointerCount()* method of the current MotionEvent object. The ID for a pointer at a particular index in the list of current pointers may be obtained via a call to the MotionEvent *getPointerId()* method. For example, the following code excerpt obtains a count of pointers and the ID of the pointer at index 0:

```
public boolean onTouch(View v, MotionEvent m) {
       int pointerCount = m.getPointerCount();
       int pointerId = m.getPointerId(0);
       return true;
}
```

Note that the pointer count will always be greater than or equal to 1 when an *onTouch()* method is called (since at least one touch must have occurred for the callback to be triggered).

A touch on a view, particularly one involving motion across the screen, will generate a stream of events before the point of contact with the screen is lifted. As such, it is likely that an application will need to track individual touches over multiple touch events. While the ID of a specific touch gesture will not change from one event to the next, it is important to keep in mind that the index value will change as other touch events come and go. When working with a touch gesture over multiple events, therefore, it is essential that the ID value be used as the touch reference in order to make sure the same touch is being tracked. When calling methods that require an index value, this should be obtained by converting the ID for a touch to the corresponding index value via a call to the *findPointerIndex()* method of the *MotionEvent* object.

26.5 **An Example Multi-Touch Application**

The example application created in the remainder of this chapter will track up to two touch gestures as they move across a layout view. As the events for each touch are triggered, the coordinates, index and ID for each touch will be displayed on the screen.

Create a new project in Android Studio, entering *MotionEvent* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *MotionEventActivity* with a corresponding layout file named *activity_motion_event.*

Click on the *Finish* button to initiate the project creation process.

26.6 **Designing the Activity User Interface**

The user interface for the application's sole activity is to consist of a ConstraintLayout view containing two TextView objects. Within the Project tool window, navigate to *app -> res -> layout* and double-click on the *activity_motion_event.xml* layout resource file to load it into the Android Studio Layout Editor tool.

Select and delete the default "Hello World!" TextView widget and then, with autoconnect enabled, drag and drop a new TextView widget so that it is centered horizontally and positioned at the 16dp margin line on the top edge of the layout:

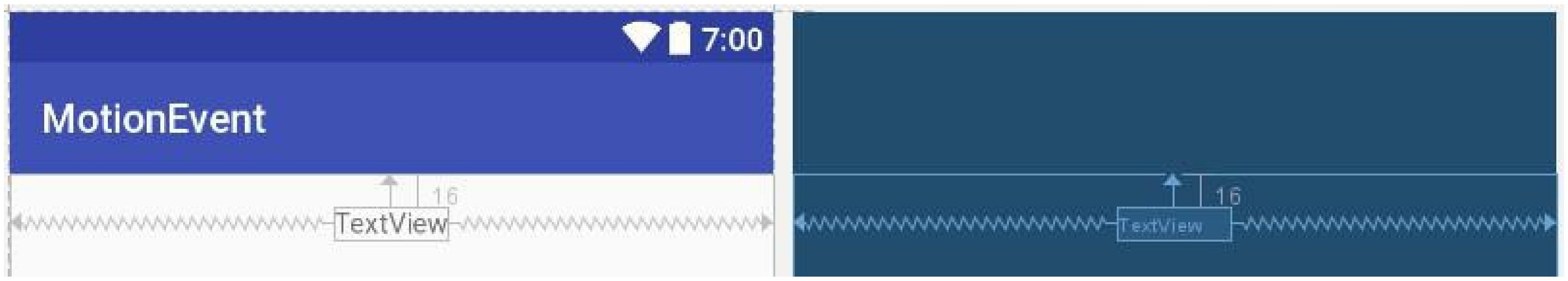


Figure 26-1

Drag a second TextView widget and position and constrain it so that it is distanced by a 32dp margin from the bottom of the first widget:

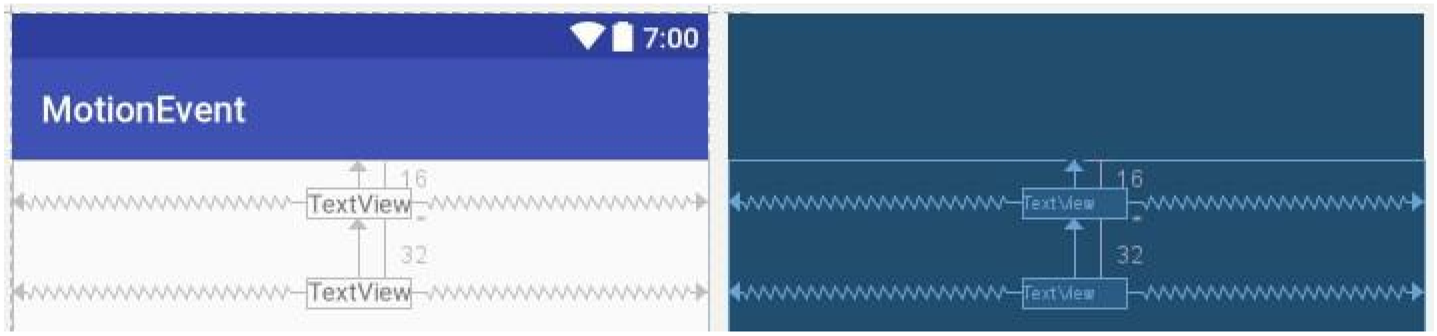


Figure 26-2

Using the Properties tool window, change the IDs for the TextView widgets to *textView1* and *textView2* respectively. Change the text displayed on the widgets to read "Touch One Status" and "Touch Two Status" and extract the strings to resources using the red warning button in the top right-hand corner of the Layout Editor.

26.7 **Implementing the Touch Event Listener**

In order to receive touch event notifications it will be necessary to register a touch listener on the layout view within the *onCreate()* method of the *MotionEventActivity* activity class. Select the *MotionEventActivity.java* tab from the Android Studio editor panel to display the source code.

Within the *onCreate()* method, add code to identify the ConstraintLayout view object, register the touch listener and implement the *onTouch()* callback method which, in this case, is going to call a second method named *handleTouch()* to which is passed the MotionEvent object:

```
package com.ebookfrenzy.motionevent;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.View;
import android.support.constraint.ConstraintLayout;
import android.widget.TextView;

public class MotionEventActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_motion_event);

        ConstraintLayout myLayout =
         (ConstraintLayout)findViewById(R.id.activity_motion_event);

        myLayout.setOnTouchListener(
                new ConstraintLayout.OnTouchListener() {
                    public boolean onTouch(View v,
                                     MotionEvent m) {
                        handleTouch(m);
                        return true;
                    }
                }
        );
   }
.
.
.
}
```

The final task before testing the application is to implement the *handleTouch()* method called by the *onTouch()* callback method. The code for this method reads as follows:

```
void handleTouch(MotionEvent m)
{
       TextView textView1 = (TextView)findViewById(R.id.textView1);
       TextView textView2 = (TextView)findViewById(R.id.textView2);

       int pointerCount = m.getPointerCount();

       for (int i = 0; i < pointerCount; i++)
       {
              int x = (int) m.getX(i);
              int y = (int) m.getY(i);
              int id = m.getPointerId(i);
              int action = m.getActionMasked();
               int actionIndex = m.getActionIndex();
              String actionString;
```

```
                switch (action)
                {
                        case MotionEvent.ACTION_DOWN:
                                actionString = "DOWN";
                                break;
                        case MotionEvent.ACTION_UP:
                                actionString = "UP";
                               break;
                        case MotionEvent.ACTION_POINTER_DOWN:
                                actionString = "PNTR DOWN";
                                break;
                        case MotionEvent.ACTION_POINTER_UP:
                                actionString = "PNTR UP";
                                break;
                        case MotionEvent.ACTION_MOVE:
                                actionString = "MOVE";
                                break;
                        default:
                                actionString = "";
                }

                String touchStatus = "Action: " + actionString + " Index: " +
    actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

                if (id == 0)
                        textView1.setText(touchStatus);
                else
                        textView2.setText(touchStatus);
        }
    }
```

Before compiling and running the application, it is worth taking the time to walk through this code systematically to highlight the tasks that are being performed.

The code begins by obtaining references to the two TextView objects in the user interface and identifying how many pointers are currently active on the view:

```
    TextView textView1 = (TextView)findViewById(R.id.textView1);
    TextView textView2 = (TextView)findViewById(R.id.textView2);

    int pointerCount = m.getPointerCount();
```

Next, the *pointerCount* variable is used to initiate a *for* loop which performs a set of tasks for each active pointer. The first few lines of the loop obtain the X and Y coordinates of the touch together with the corresponding event ID, action type and action index. Lastly, a string variable is declared:

```
    for (int i = 0; i < pointerCount; i++)
    {
                int x = (int) m.getX(i);
                int y = (int) m.getY(i);
                int id = m.getPointerId(i);
                int action = m.getActionMasked();
                int actionIndex = m.getActionIndex();
                String actionString;
```

Since action types equate to integer values, a *switch* statement is used to convert the action type to a more meaningful string value, which is stored in the previously declared *actionString* variable:

```
    switch (action)
                  {
                          case MotionEvent.ACTION_DOWN:
                                  actionString = "DOWN";
                                  break;
                          case MotionEvent.ACTION_UP:
                                  actionString = "UP";
                                  break;
                          case MotionEvent.ACTION_POINTER_DOWN:
                                  actionString = "PNTR DOWN";
                                  break;
                          case MotionEvent.ACTION_POINTER_UP:
                                  actionString = "PNTR UP";
                                  break;
                          case MotionEvent.ACTION_MOVE:
                                  actionString = "MOVE";
                                  break;
                          default:
                                  actionString = "";
                  }
```

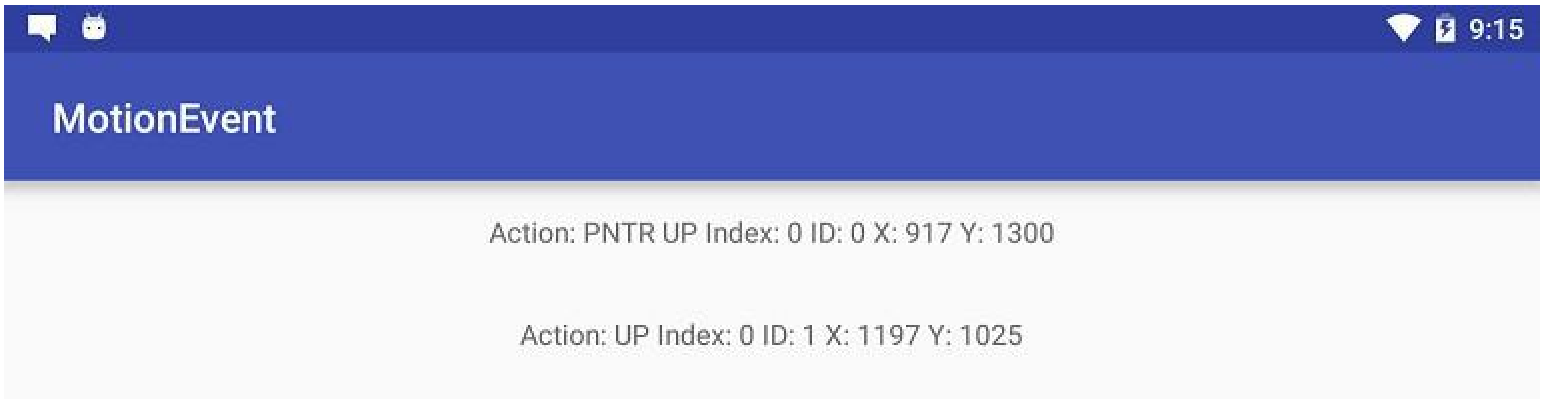
Lastly, the string message is constructed using the *actionString* value, the action index, touch ID and X and Y coordinates. The ID value is then used to decide whether the string should be displayed on the first or second TextView object:

```
    String touchStatus = "Action: " + actionString + " Index: "
             + actionIndex + " ID: " + id + " X: " + x + " Y: " + y;

                  if (id == 0)
                          textView1.setText(touchStatus);
                   else
                          textView2.setText(touchStatus);
```

26.8 **Running the Example Application**

Since the Android emulator environment does not support multi-touch, compile and run the application on a physical Android device. Once launched, experiment with single and multiple touches on the screen and note that the text views update to reflect the events as illustrated in Figure 26-3:



Action: PNTR UP Index: 0 ID: 0 X: 917 Y: 1300

Action: UP Index: 0 ID: 1 X: 1197 Y: 1025

Figure 26-3

26.9 **Summary**

Activities receive notifications of touch events by registering an onTouchListener event listener and implementing the *onTouch()* callback method which, in turn, is passed a MotionEvent object when called by the Android runtime. This object contains information about the touch such as the type of touch event, the coordinates of the touch and a count of the number of touches currently in contact with the view.

When multiple touches are involved, each point of contact is referred to as a pointer with each assigned an index and an ID. While the index of a touch can change from one event to another, the ID will remain unchanged until the touch ends.

This chapter has worked through the creation of an example Android application designed to display the coordinates and action type of up to two simultaneous touches on a device display.

Having covered touches in general, the next chapter (entitled *Detecting Common Gestures using the Android Gesture Detector Class*) will look further at touch screen event handling through the implementation of gesture recognition.

27. Detecting Common Gestures using the Android Gesture Detector Class

The term "gesture" is used to define a contiguous sequence of interactions between the touch screen and the user. A typical gesture begins at the point that the screen is first touched and ends when the last finger or pointing device leaves the display surface. When correctly harnessed, gestures can be implemented as a form of communication between user and application. Swiping motions to turn the pages of an eBook, or a pinching movement involving two touches to zoom in or out of an image are prime examples of the ways in which gestures can be used to interact with an application.

The Android SDK provides mechanisms for the detection of both common and custom gestures within an application. Common gestures involve interactions such as a tap, double tap, long press or a swiping motion in either a horizontal or a vertical direction (referred to in Android nomenclature as a *fling*).

The goal of this chapter is to explore the use of the Android GestureDetector class to detect common gestures performed on the display of an Android device. The next chapter, entitled *Implementing Custom Gesture and Pinch Recognition on Android*, will cover the detection of more complex, custom gestures such as circular motions and pinches.

27.1 Implementing Common Gesture Detection

When a user interacts with the display of an Android device, the *onTouchEvent()* method of the currently active application is called by the system and passed MotionEvent objects containing data about the user's contact with the screen. This data can be interpreted to identify if the motion on the screen matches a common gesture such as a tap or a swipe. This can be achieved with very little programming effort by making use of the Android GestureDetectorCompat class. This class is designed specifically to receive motion event information from the application and to trigger method calls based on the type of common gesture, if any, detected.

The basic steps in detecting common gestures are as follows:

1. Declaration of a class which implements the GestureDetector.OnGestureListener interface including the required *onFling()*, *onDown()*,*onScroll()*, *onShowPress()*, *onSingleTapUp()* and *onLongPress()* callback methods. Note that this can be either an entirely new class, or the enclosing activity class. In the event that double tap gesture detection is required, the class must also implement the GestureDetector.OnDoubleTapListener interface and include the corresponding *onDoubleTap()* method.
2. Creation of an instance of the Android GestureDetectorCompat class, passing through an instance of the class created in step 1 as an argument.
3. An optional call to the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance to enable double tap detection if required.
4. Implementation of the *onTouchEvent()* callback method on the enclosing activity which, in turn, must call the *onTouchEvent()* method of the GestureDetectorCompat instance, passing through the current motion event object as an argument to the method.

Once implemented, the result is a set of methods within the application code that will be called when a gesture of a particular type is detected. The code within these methods can then be implemented to perform any tasks that need to be performed in response to the corresponding gesture.

In the remainder of this chapter, we will work through the creation of an example project intended to put the above steps into practice.

27.2 **Creating an Example Gesture Detection Project**

The goal of this project is to detect the full range of common gestures currently supported by the GestureDetectorCompat class and to display status information to the user indicating the type of gesture that has been detected.

Create a new project in Android Studio, entering *CommonGestures* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CommonGesturesActivity* with a corresponding layout resource file named *activity_common_gestures*.

Click on the *Finish* button to initiate the project creation process.

Once the new project has been created, navigate to the *app -> res -> layout -> activity_common_gestures.xml* file in the Project tool window and double-click on it to load it into the Layout Editor tool.

Within the Layout Editor tool, select the “Hello, World!” TextView component and, in the Properties tool window, enter *gestureStatusText* as the ID.

27.3 **Implementing the Listener Class**

As previously outlined, it is necessary to create a class that implements the GestureDetector.OnGestureListener interface and, if double tap detection is required, the GestureDetector.OnDoubleTapListener interface. While this can be an entirely new class, it is also perfectly valid to implement this within the current activity class. For the purposes of this example, therefore, we will modify the CommonGesturesActivity class to implement these listener interfaces. Edit the *CommonGesturesActivity.java* file so that it reads as follows to declare the interfaces and to extract and store a reference to the TextView component in the user interface:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;

public class CommonGesturesActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener
{
    private TextView gestureText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);

        gestureText =
                (TextView)findViewById(R.id.gestureStatusText);

    }
.
.
.
}
```

Declaring that the class implements the listener interfaces mandates that the corresponding methods also be implemented in the class:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;
import android.view.MotionEvent;


public class CommonGesturesActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener {

    private TextView gestureText;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);
        gestureText =
                (TextView)findViewById(R.id.gestureStatusText);
    }

    @Override
    public boolean onDown(MotionEvent event) {
        gestureText.setText ("onDown");
        return true;
    }

    @Override
    public boolean onFling(MotionEvent event1, MotionEvent event2,
                           float velocityX, float velocityY) {
        gestureText.setText("onFling");
        return true;
    }

    @Override
    public void onLongPress(MotionEvent event) {
        gestureText.setText("onLongPress");
    }
    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2,
                            float distanceX, float distanceY) {
        gestureText.setText("onScroll");
        return true;
    }

    @Override
    public void onShowPress(MotionEvent event) {
        gestureText.setText("onShowPress");
    }

    @Override
    public boolean onSingleTapUp(MotionEvent event) {
        gestureText.setText("onSingleTapUp");
        return true;
    }

    @Override
    public boolean onDoubleTap(MotionEvent event) {
        gestureText.setText("onDoubleTap");
        return true;
    }

    @Override
    public boolean onDoubleTapEvent(MotionEvent event) {
        gestureText.setText("onDoubleTapEvent");
        return true;
    }

    @Override
    public boolean onSingleTapConfirmed(MotionEvent event) {
        gestureText.setText("onSingleTapConfirmed");
        return true;
    }
.
.
.
}
```

Note that many of these methods return *true.* This indicates to the Android Framework that the event has been consumed by the method and does not need to be passed to the next event handler in the stack.

27.4 **Creating the GestureDetectorCompat Instance**

With the activity class now updated to implement the listener interfaces, the next step is to create an instance of the GestureDetectorCompat class. Since this only needs to be performed once at the point that the activity is created, the best place for this code is in the *onCreate()* method. Since we also want to detect double taps, the code also needs to call the *setOnDoubleTapListener()* method of the GestureDetectorCompat instance:

```
package com.ebookfrenzy.commongestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.GestureDetector;
import android.widget.TextView;
import android.view.MotionEvent;
import android.support.v4.view.GestureDetectorCompat;

public class CommonGesturesActivity extends AppCompatActivity
        implements GestureDetector.OnGestureListener,
        GestureDetector.OnDoubleTapListener {

    private TextView gestureText;
    private GestureDetectorCompat gDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_common_gestures);
        gestureText =
                (TextView)findViewById(R.id.gestureStatusText);

        this.gDetector = new GestureDetectorCompat(this,this);
        gDetector.setOnDoubleTapListener(this);
    }
.
.
}
```

27.5 **Implementing the onTouchEvent() Method**

If the application were to be compiled and run at this point, nothing would happen if gestures were performed on the device display. This is because no code has been added to intercept touch events and to pass them through to the GestureDetectorCompat instance. In order to achieve this, it is necessary to override the *onTouchEvent()* method within the activity class and implement it such that it calls the *onTouchEvent()* method of the GestureDetectorCompat instance. Remaining in the *CommonGesturesActivity.java* file, therefore, implement this method so that it reads as follows:

```
@Override
public boolean onTouchEvent(MotionEvent event) {
        this.gDetector.onTouchEvent(event);
        // Be sure to call the superclass implementation
        return super.onTouchEvent(event);
}
```

27.6 **Testing the Application**

Compile and run the application on either a physical Android device or an AVD emulator. Once launched, experiment with swipes, presses, scrolling motions and double and single taps. Note that the text view updates to reflect the events as illustrated in Figure 27-1:

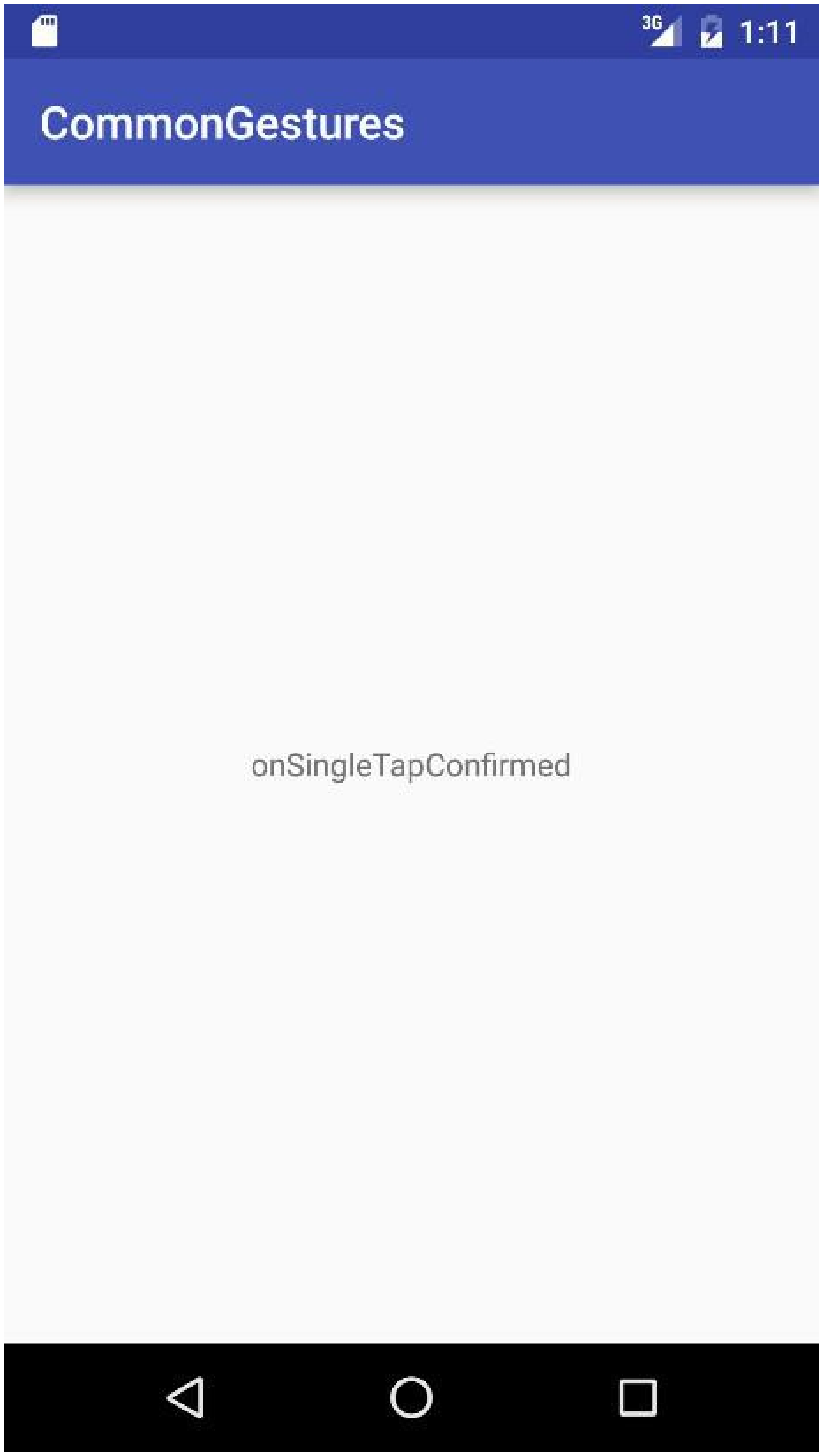


Figure 27-1

27.7 Summary

Any physical contact between the user and the touch screen display of a device can be considered a “gesture”. Lacking the physical keyboard and mouse pointer of a traditional computer system, gestures are widely used as a method of interaction between user and application. While a gesture can be comprised of just about any sequence of motions, there is a widely used set of gestures with which users of touch screen devices have become familiar. A number of these so-called “common gestures” can be easily detected within an application by making use of the Android Gesture Detector classes. In this chapter, the use of this technique has been outlined both in theory and through the implementation of an example project.

Having covered common gestures in this chapter, the next chapter will look at detecting a wider range of gesture types including the ability to both design and detect your own gestures.

28. Implementing Custom Gesture and Pinch Recognition on Android

The previous chapter looked at the steps involved in detecting what are referred to as “common gestures” from within an Android application. In practice, however, a gesture can conceivably involve just about any sequence of touch motions on the display of an Android device. In recognition of this fact, the Android SDK allows custom gestures of just about any nature to be defined by the application developer and used to trigger events when performed by the user. This is a multistage process, the details of which are the topic of this chapter.

28.1 The Android Gesture Builder Application

The Android SDK allows developers to design custom gestures which are then stored in a gesture file bundled with an Android application package. These custom gesture files are most easily created using the *Gesture Builder* application which is bundled with the samples package supplied as part of the Android SDK. The creation of a gestures file involves launching the Gesture Builder application, either on a physical device or emulator, and “drawing” the gestures that will need to be detected by the application. Once the gestures have been designed, the file containing the gesture data can be pulled off the SD card of the device or emulator and added to the application project. Within the application code, the file is then loaded into an instance of the *GestureLibrary* class where it can be used to search for matches to any gestures performed by the user on the device display.

28.2 The GestureOverlayView Class

In order to facilitate the detection of gestures within an application, the Android SDK provides the GestureOverlayView class. This is a transparent view that can be placed over other views in the user interface for the sole purpose of detecting gestures.

28.3 Detecting Gestures

Gestures are detected by loading the gestures file created using the Gesture Builder app and then registering a *GesturePerformedListener* event listener on an instance of the GestureOverlayView class. The enclosing class is then declared to implement both the *OnGesturePerformedListener* interface and the corresponding *onGesturePerformed* callback method required by that interface. In the event that a gesture is detected by the listener, a call to the *onGesturePerformed* callback method is triggered by the Android runtime system.

28.4 Identifying Specific Gestures

When a gesture is detected, the *onGesturePerformed* callback method is called and passed as arguments a reference to the GestureOverlayView object on which the gesture was detected, together with a Gesture object containing information about the gesture.

With access to the Gesture object, the GestureLibrary can then be used to compare the detected gesture to those contained in the gestures file previously loaded into the application. The GestureLibrary reports the probability that the gesture performed by the user matches an entry in the gestures file by calculating a *prediction score* for each gesture. A prediction score of 1.0 or greater is generally accepted to be a good match between a gesture stored in the file and that performed by the user on the device display.

28.5 **Building and Running the Gesture Builder Application**

The Gesture Builder application is bundled by default with the AVD emulator profile for most versions of the SDK. It is not, however, pre-installed on most physical Android devices. If the utility is pre-installed, it will be listed along with the other apps installed in the device or AVD instance. In the event that it is not installed, the source code for the utility is included with the sample code provided with this book. If you have not already done so, download this now using the following link:

http://www.ebookfrenzy.com/retail/androidstudio23/index.php

The source code for the Gesture Builder application is located within this archive in a folder named *GestureBuilder*.

The GestureBuilder project is based on Android 5.0.1 (API 21) so use the SDK Manager tool once again to ensure that this version of the Android SDK is installed before proceeding.

From the Android Studio welcome screen select the *Import project* option. Alternatively, from the Android Studio main window for an existing project, select the *File -> New -> Import Project…* menu option and, within the resulting dialog, navigate to and select the GestureBuilder folder within the samples directory and click on *OK*. At this point, Android Studio will import the project into the designated folder and convert it to match the Android Studio project file and build structure.

Once imported, install and run the GestureBuilder utility on an Android device attached to the development system.

28.6 **Creating a Gestures File**

Once the Gesture Builder application has loaded, it should indicate that no gestures have yet been created. To create a new gesture, click on the *Add gesture* button located at the bottom of the device screen, enter the name *Circle Gesture* into the *Name* text box and then “draw” a gesture using a circular motion on the screen as illustrated in Figure 28-1. Assuming that the gesture appears as required (represented by the yellow line on the device screen), click on the *Done* button to add the gesture to the gestures file:

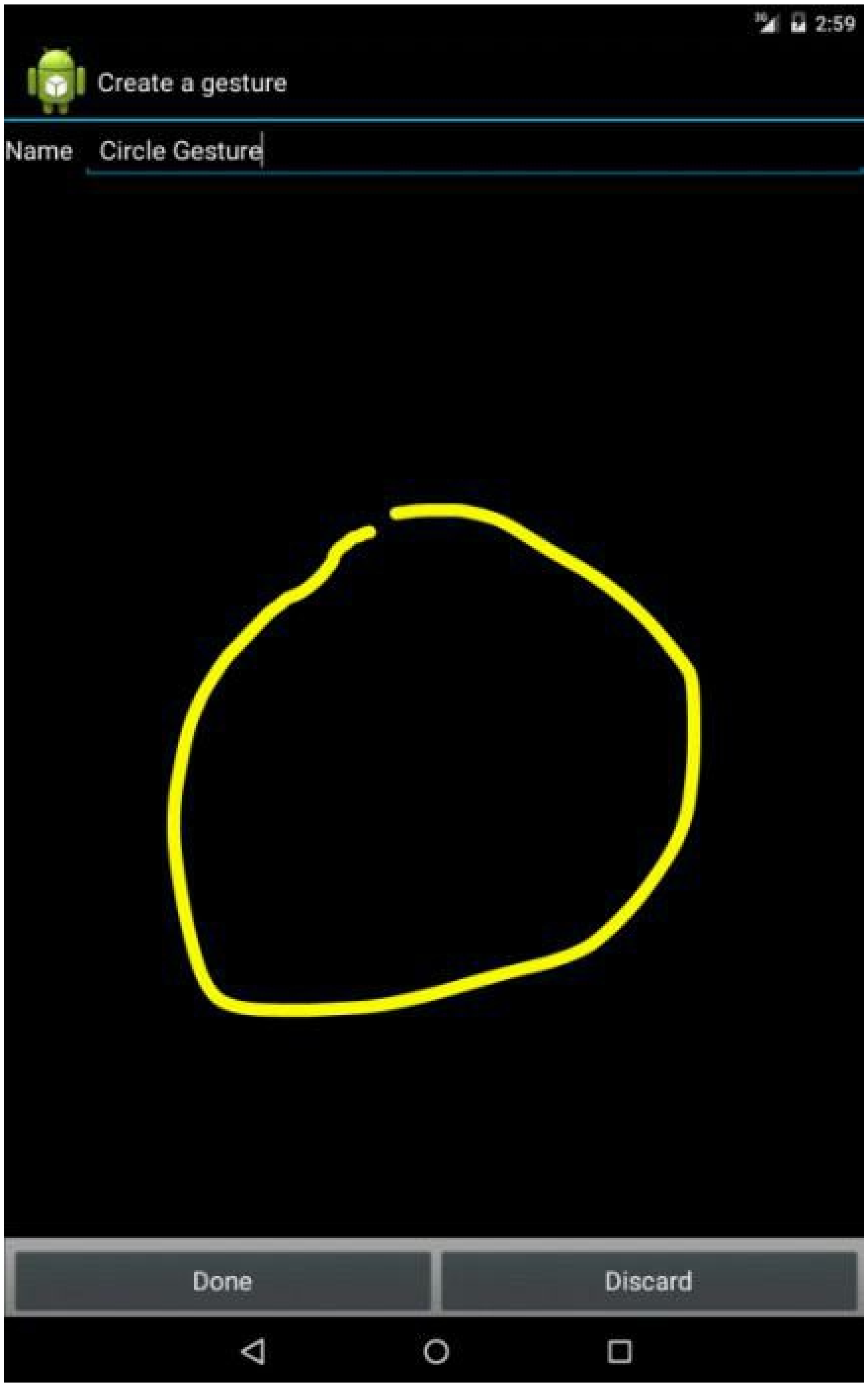


Figure 28-1

After the gesture has been saved, the Gesture Builder app will display a list of currently defined gestures, which, at this point, will consist solely of the new *Circle Gesture*.

Repeat the gesture creation process to add a further gesture to the file. This should involve a two-stroke gesture creating an X on the screen named *X Gesture*. When creating gestures involving multiple strokes, be sure to allow as little time as possible between each stroke so that the builder knows that the strokes are part of the same gesture. Once this gesture has been added, the list within the Gesture Builder application should resemble that outlined in Figure 28-2:

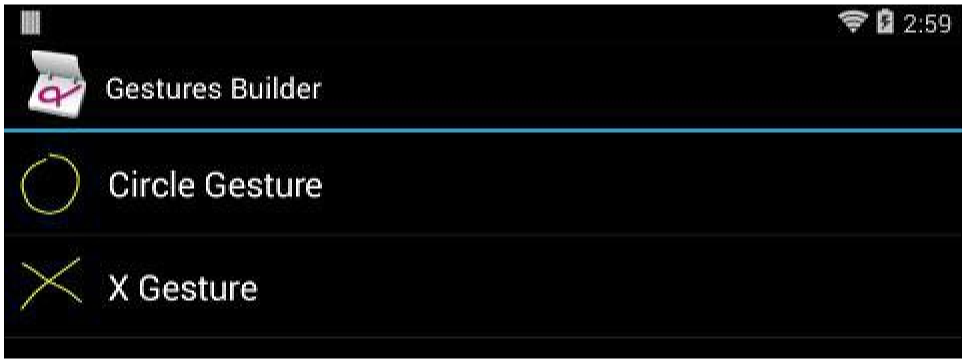


Figure 28-2

28.7 Extracting the Gestures File from the SD Card

As each gesture was created within the Gesture Builder application, it was added to a file named *gestures* located on the SD Card of the emulator or device on which the app was running. Before this file can be added to an Android Studio project, however, it must first be pulled off the SD Card and saved to the local file system. This is most easily achieved by using the *adb* command-line tool. Open a Terminal or Command Prompt window, therefore, and execute the following command:

```
adb devices
```

In the event that the adb command is not found, refer to *Setting up an Android Studio Development Environment* for guidance on adding this to the PATH environment variable of your system.

Once executed, the command will list all active physical devices and AVD instances attached to the system. The following output, for example, indicates that both a physical device and one AVD emulator have been detected on the development computer system:

```
List of devices attached
HT4CTJT01906    device
emulator-5554   device
```

In order to pull the gestures file from the emulator in the above example and place it into the current working directory of the Terminal or Command Prompt window, the following command would need to be executed:

```
adb -s emulator-5554 pull /sdcard/gestures .
```

Alternatively, the gestures file can be pulled from a device connected via adb using the following command (where the –d flag is used to indicate a physical device):

```
adb -d pull /sdcard/gestures .
```

Once the gestures file has been created and pulled off the SD Card, it is ready to be added to an Android Studio project as a resource file. The next step, therefore, is to create a new project.

28.8 Creating the Example Project

Create a new project in Android Studio, entering *CustomGestures* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CustomGesturesActivity* with a corresponding layout file named *activity_custom_gestures*.

Click on the *Finish* button to initiate the project creation process.

28.9 **Adding the Gestures File to the Project**

Within the Android Studio Project tool window, locate and right-click on the *res* folder (located under *app*) and select *New -> Directory* from the resulting menu. In the New Directory dialog, enter *raw* as the folder name and click on the *OK* button. Using the appropriate file explorer utility for your operating system type, locate the *gestures* file previously pulled from the SD Card and copy and paste it into the new *raw* folder in the Project tool window.

28.10 **Designing the User Interface**

This example application calls for a very simple user interface consisting of a LinearLayout view with a GestureOverlayView layered on top of it to intercept any gestures performed by the user. Locate the *app -> res -> layout -> activity_custom_gestures.xml* file and double-click on it to load it into the Layout Editor tool.

Once loaded, switch to Text mode and modify the XML so that it reads as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal">

    </android.gesture.GestureOverlayView>
</LinearLayout>
```

28.11 **Loading the Gestures File**

Now that the gestures file has been added to the project, the next step is to write some code so that the file is loaded when the activity starts up. For the purposes of this project, the code to achieve this will be placed in the *onCreate()* method of the *CustomGesturesActivity* class located in the *CustomGesturesActivity.java* source file as follows:

```
package com.ebookfrenzy.customgestures;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;

public class CustomGesturesActivity extends AppCompatActivity
        implements OnGesturePerformedListener {

    private GestureLibrary gLibrary;
```

```
        @Override
        protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_custom_gestures);

            gLibrary =
                    GestureLibraries.fromRawResource(this,
                         R.raw.gestures);
            if (!gLibrary.load()) {
                finish();
            }
        }
    .
    .
    .
    }
```

In addition to some necessary import directives, the above code changes to the *onCreate()* method also create a *GestureLibrary* instance named *gLibrary* and then loads into it the contents of the gestures file located in the *raw* resources folder. The activity class has also been modified to implement the *OnGesturePerformedListener* interface, which requires the implementation of the *onGesturePerformed* callback method (which will be created in a later section of this chapter).

28.12 **Registering the Event Listener**

In order for the activity to receive notification that the user has performed a gesture on the screen, it is necessary to register the *OnGesturePerformedListener* event listener on the *gLayout* view, a reference to which can be obtained using the *findViewById* method as outlined in the following code fragment:

```
    @Override
    protected void onCreate(Bundle savedInstanceState) {
           super.onCreate(savedInstanceState);
           setContentView(R.layout.activity_custom_gestures);

           gLibrary =
                 GestureLibraries.fromRawResource(this, R.raw.gestures);
          if (!gLibrary.load()) {
                finish();
          }

          GestureOverlayView gOverlay =
                    (GestureOverlayView) findViewById(R.id.gOverlay);
          gOverlay.addOnGesturePerformedListener(this);
    }
```

28.13 **Implementing the onGesturePerformed Method**

All that remains before an initial test run of the application can be performed is to implement the *OnGesturePerformed* callback method. This is the method which will be called when a gesture is performed on the GestureOverlayView instance:

```
    package com.ebookfrenzy.customgestures;
```

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.gesture.GestureLibraries;
import android.gesture.GestureLibrary;
import android.gesture.GestureOverlayView;
import android.gesture.GestureOverlayView.OnGesturePerformedListener;
import android.gesture.Prediction;
import android.widget.Toast;
import android.gesture.Gesture;
import java.util.ArrayList;

public class CustomGesturesActivity extends AppCompatActivity implements
OnGesturePerformedListener {

    private GestureLibrary gLibrary;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_custom_gestures);

        gLibrary =
                GestureLibraries.fromRawResource(this,
                      R.raw.gestures);
        if (!gLibrary.load()) {
            finish();
        }

        GestureOverlayView gOverlay =
                (GestureOverlayView) findViewById(R.id.gOverlay);
        gOverlay.addOnGesturePerformedListener(this);
    }

    public void onGesturePerformed(GestureOverlayView overlay, Gesture
            gesture) {
        ArrayList<Prediction> predictions =
                gLibrary.recognize(gesture);

        if (predictions.size() > 0 && predictions.get(0).score > 1.0)
        {

            String action = predictions.get(0).name;

            Toast.makeText(this, action, Toast.LENGTH_SHORT).show();
        }
    }
.
.
.
}
```

When a gesture on the gesture overlay view object is detected by the Android runtime, the *onGesturePerformed* method is called. Passed through as arguments are a reference to the GestureOverlayView object on which the gesture was detected together with an object of type *Gesture*. The Gesture class is designed to hold the information that defines a specific gesture (essentially a sequence of timed points on the screen depicting the path of the strokes that comprise a gesture).

The Gesture object is passed through to the *recognize()* method of our *gLibrary* instance, the purpose of which is to compare the current gesture with each gesture loaded from the gestures file. Once this task is complete, the *recognize()* method returns an ArrayList object containing a Prediction object for each comparison performed. The list is ranked in order from the best match (at position 0 in the array) to the worst. Contained within each prediction object is the name of the corresponding gesture from the gestures file and a prediction score indicating how closely it matches the current gesture.

The code in the above method, therefore, takes the prediction at position 0 (the closest match) makes sure it has a score of greater than 1.0 and then displays a Toast message (an Android class designed to display notification pop ups to the user) displaying the name of the matching gesture.

28.14 **Testing the Application**

Build and run the application on either an emulator or a physical Android device and perform the circle and swipe gestures on the display. When performed, the toast notification should appear containing the name of the gesture that was performed. Note that when a gesture is recognized, it is outlined on the display with a bright yellow line while gestures about which the overlay is uncertain appear as a faded yellow line. While useful during development, this is probably not ideal for a real world application. Clearly, therefore, there is still some more configuration work to do.

28.15 **Configuring the GestureOverlayView**

By default, the GestureOverlayView is configured to display yellow lines during gestures. The color used to draw recognized and unrecognized gestures can be defined via the *android:gestureColor* and *android:uncertainGestureColor* properties. For example, to hide the gesture lines, modify the *activity_custom_gestures.xml* file in the example project as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    xmlns:android="http://schemas.android.com/apk/res/android">

    <android.gesture.GestureOverlayView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/gOverlay"
        android:layout_gravity="center_horizontal"
        android:gestureColor="#00000000"
        android:uncertainGestureColor="#00000000" >
    </android.gesture.GestureOverlayView>
</LinearLayout>
```

On re-running the application, gestures should now be invisible (since they are drawn in white on the white background of the LinearLayout view).

28.16 **Intercepting Gestures**

The GestureOverlayView is, as previously described, a transparent overlay that may be positioned over the top of other views. This leads to the question as to whether events intercepted by the gesture overlay should then be passed on to the underlying views when a gesture has been recognized. This is controlled via the *android:eventsInterceptionEnabled* property of the GestureOverlayView instance. When set to true, the gesture events are not passed to the underlying views when a gesture is recognized. This can be a particularly useful setting when gestures are being performed over a view that might be configured to scroll in response to certain gestures. Setting this property to *true* will avoid gestures also being interpreted as instructions to the underlying view to scroll in a particular direction.

28.17 **Detecting Pinch Gestures**

Before moving on from touch handling in general and gesture recognition in particular, the last topic of this chapter is that of handling pinch gestures. While it is possible to create and detect a wide range of gestures using the steps outlined in the previous sections of this chapter it is, in fact, not possible to detect a pinching gesture (where two fingers are used in a stretching and pinching motion, typically to zoom in and out of a view or image) using the techniques discussed so far.

The simplest method for detecting pinch gestures is to use the Android *ScaleGestureDetector* class. In general terms, detecting pinch gestures involves the following three steps:

1. Declaration of a new class which implements the SimpleOnScaleGestureListener interface including the required *onScale()*, *onScaleBegin()* and *onScaleEnd()* callback methods.
2. Creation of an instance of the ScaleGestureDetector class, passing through an instance of the class created in step 1 as an argument.
3. Implementing the *onTouchEvent()* callback method on the enclosing activity which, in turn, calls the *onTouchEvent()* method of the ScaleGestureDetector class.

In the remainder of this chapter, we will create a very simple example designed to demonstrate the implementation of pinch gesture recognition.

28.18 **A Pinch Gesture Example Project**

Create a new project in Android Studio, entering *PinchExample* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 14: Android 4.0 (IceCreamSandwich). Continue to proceed through the screens, requesting the creation of an Empty Activity named *PinchExampleActivity* with a layout resource file named *activity_pinch_example.*

Within the *activity_pinch_example.xml* file, select the default TextView object and use the Properties tool window to set the ID to *myTextView*.

Locate and load the *PinchExampleActivity.java* file into the Android Studio editor and modify the file as follows:

```
package com.ebookfrenzy.pinchexample;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.view.MotionEvent;
import android.view.ScaleGestureDetector;
import android.view.ScaleGestureDetector.SimpleOnScaleGestureListener;
import android.widget.TextView;

public class PinchExampleActivity extends AppCompatActivity {

    TextView scaleText;
    ScaleGestureDetector scaleGestureDetector;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pinch_example);

        scaleText = (TextView)findViewById(R.id.myTextView);

        scaleGestureDetector =
                new ScaleGestureDetector(this,
                        new MyOnScaleGestureListener());
    }

    @Override
    public boolean onTouchEvent(MotionEvent event) {
        scaleGestureDetector.onTouchEvent(event);
        return true;
    }

    public class MyOnScaleGestureListener extends
            SimpleOnScaleGestureListener {

        @Override
        public boolean onScale(ScaleGestureDetector detector) {

            float scaleFactor = detector.getScaleFactor();

            if (scaleFactor > 1) {
                scaleText.setText("Zooming Out");
            } else {
                scaleText.setText("Zooming In");
            }
            return true;
        }

        @Override
        public boolean onScaleBegin(ScaleGestureDetector detector) {
            return true;
        }

        @Override
        public void onScaleEnd(ScaleGestureDetector detector) {

        }
    }
.
.
.
}
```

The code begins by declaring TextView and ScaleGestureDetector variables. A new class named MyOnScaleGestureListener is declared which extends the Android SimpleOnScaleGestureListener class. This interface requires that three methods (*onScale()*, *onScaleBegin()* and *onScaleEnd()*) be implemented. In this instance the *onScale()* method identifies the scale factor and displays a message on the text view indicating the type of pinch gesture detected.

Within the *onCreate()* method, a reference to the text view object is obtained and assigned to the scaleText variable. Next, a new *ScaleGestureDetector* instance is created, passing through a reference to the enclosing activity and an instance of our new *MyOnScaleGestureListener* class as arguments. Finally, an *onTouchEvent()* callback method is implemented for the activity, which simply calls the corresponding *onTouchEvent()* method of the *ScaleGestureDetector* object, passing through the MotionEvent object as an argument.

Compile and run the application on an emulator or physical Android device and perform pinching gestures on the screen, noting that the text view displays either the zoom in or zoom out message depending on the pinching motion. Pinching gestures may be simulated within the emulator by holding down the Ctrl key and clicking and dragging the mouse pointer as shown in Figure 28-3:

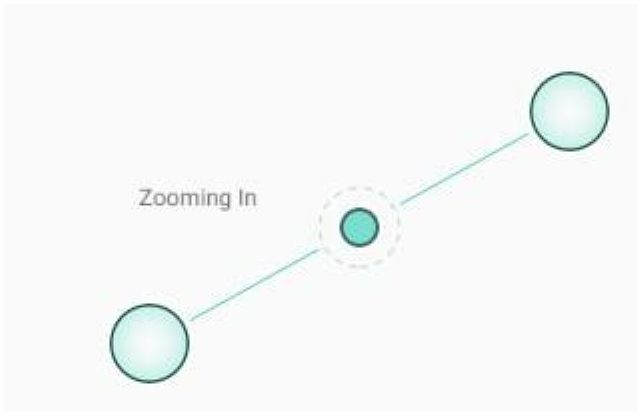


Figure 28-3

28.19 **Summary**

A gesture is essentially the motion of points of contact on a touch screen involving one or more strokes and can be used as a method of communication between user and application. Android allows gestures to be designed using the Gesture Builder application. Once created, gestures can be saved to a gestures file and loaded into an activity at application runtime using the GestureLibrary.

Gestures can be detected on areas of the display by overlaying existing views with instances of the transparent *GestureOverlayView* class and implementing an *OnGesturePerformedListener* event listener. Using the GestureLibrary, a ranked list of matches between a gesture performed by the user and the gestures stored in a gestures file may be generated, using a prediction score to decide whether a gesture is a close enough match.

Pinch gestures may be detected through the implementation of the ScaleGestureDetector class, an example of which was also provided in this chapter.